UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Uma Estratégia para Implementar Refatorações Seguras no Eclipse

Trabalho de Conclusão de Curso

Igor Nascimento dos Santos

São Cristóvão – Sergipe

2018

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Igor Nascimento dos Santos

Uma Estratégia para Implementar Refatorações Seguras no Eclipse

Trabalho de Conclusão de Curso submetido ao Departamento de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador(a): Giovanny Fernando Lucero Palma

São Cristóvão – Sergipe

2018

Dedico este trabalho aos meus pais, familiares e amigos que me deram o suporte necessário para realizá-lo.

Agradecimentos

Agradeço primeiramente à minha mãe Maria Lourdes do Nascimento por nunca medir esforços para me fazer feliz. Perfeita, em cada um dos seus defeitos, ensinou-me a sempre dar o melhor de mim quando o assunto é fazer o bem ao próximo. Espero um dia retribuir tudo que ela já me fez, faz e continuará fazendo por mim. Este trabalho, toda a graduação, cursos, empregos e cada boa ação que eu fizer na vida é consequência dela.

Agradeço ao meu pai José Messias dos Santos por ser um exemplo para minha vida e estar sempre disposto a me ajudar e me guiar pelo caminho do bem e da honestidade, além de sempre estar presente em minha vida. Este trabalho reflete também o seu esforço em buscar a evolução, não somente própria, mas familiar.

Agradeço à minha irmã Alice Cristina Nascimento Alves por todo incentivo – muitas vezes em formas de discussões e brigas – que me impulsionou a chegar tão longe. A diferença de idade, de gostos e inúmeras distinções entre nós nunca nos separou, ao contrário, nos fez unidos em aspectos que não conseguimos nem verbalizar.

Agradeço a toda família Nascimento, por todo apoio e diversão promovida. Vocês são a prova real de que a melhor coisa na vida é ter a família presente.

Agradeço também aos professores e funcionários do colégio João Batista da Rocha. Foi lá onde cultivei amigos e adquiri conhecimento para cursar toda a minha graduação.

Agradeço aos meus professores da UFS, em especial àqueles que me orientaram de alguma forma. Em minha primeira iniciação científica, o professor Henrique Nou Schneider despertou em mim o desejo em contribuir com a ciência com inovação e disciplina. Orientado-me neste trabalho e na minha segunda iniciação científica, agradeço a Giovanny Fernando Lucero Palma pelo conhecimento e suporte fornecido, fazendo destes trabalhos os mais relevantes da minha carreira acadêmica, até então.

Agradeço a todos os funcionários do IFS por me proporcionarem um imenso amadurecimento e crescimento profissional e pessoal. Agradeço em especial a Fernando Lucas de Oliveira Farias por ser muito mais que um chefe: um líder que acreditou no meu potencial e contribuiu muito para o complemento da minha formação.

Pensamentos conduzem a sentimentos
Sentimentos conduzem a ações
Ações conduzem a resultados.
(T. Harv Eker)

Resumo

O processo de desenvolvimento de um *software* é complexo e requer processos bem definidos acerca de cada fase de trabalho. Na fase de implementação dos módulos são necessários cuidados para que o *software* obtenha índices positivos em atributos de qualidade. Um software mal implementado pode decair em atributos como manutenibilidade e reusabilidade. Um software que não é manutenível tem grandes chances de ter um curto período no mercado.

De modo a melhorar pontos problemáticos no *software*, é possível utilizar técnicas de refatoração. Refatoração de software consiste de melhorias na estrutura interna de um programa, através de transformações, mantendo o comportamento original do programa.

O catálogo de Fowler é uma das principais referências em termos de refatoração. Nele as refatorações são descritas informalmente, através de passos determinados, instruções para verificação da imutabilidade do comportamento e exemplos demonstrativos. Entretanto, a utilização de tal abordagem para refatorar pode ocasionar erros e ser muito trabalhosa, devido ao cunho manual da mesma.

Visando poupar o desenvolvedor de ter que refatorar manualmente o seu código, ferramentas de desenvolvimento como Eclipse, Netbeans e Intellij Idea automatizam refatorações similares às contidas no catálogo de Fowler. Porém, por diversos motivos, algumas refatorações disponíveis nessas ferramentas não garantem a preservação de comportamento do programa.

Estudos mostram que é possível formalizar refatorações de modo que seja possível garantir o comportamento do programa, seguindo determinadas condições. Um destes trabalhos baseia-se em leis algébricas de programação combinadas com leis para programação orientada a objetos. Utilizando essas leis, pode-se formalizar quaisquer transformações entre programas implementados em alguma linguagem orientada a objetos, como Java ou C++, e provar que o comportamento é preservado.

Observando a necessidade de que ferramentas de desenvolvimento possam oferecer refatorações confiáveis, este trabalho propõe uma estratégia para a implementação de refatorações corretas no ambiente Eclipse. Esta abordagem se fundamenta na formalização da refatoração, implementado refatorações seguindo as condições formuladas e tratando eventuais casos de *aliasing* com técnicas de análise de fluxo de dados.

Como estudo de caso, implementaremos uma versão segura da refatoração *Inline Temp* para Java, realizando correções e incrementando a versão disponível no Eclipse. As condições estáticas que a refatoração necessita são implementadas e verificas normalmente. Porém, para as condições

dinâmicas, que envolvem verificação de *aliasing*, propõe-se uma abordagem que gera obrigações de prova descritas como assertivas.

É um fato conhecido que boa parte dos desenvolvedores têm pouco contato com formalismos e refatorações formais. Desta forma, a instrumentação do código com assertivas pode ser pouco pragmática. Visando reduzir as obrigações de prova geradas pelas análises dinâmicas, este trabalho realiza um estudo sobre análise de fluxo de dados, técnicas e ferramentas que implementam análise de ponteiros. Assim, será possível não instrumentar um determinado trecho de código com assertivas, quando o mesmo não apresentar indícios de *aliasing*.

Palavras-chave: Refatoração, Eclipse, *Aliasing*, Verificação, Análise de Ponteiros.

Abstract

The process of software development is complex and requires well-defined processes about each phase of work. In the phase of implementation of the modules some care is taken so that the software obtains positive indexes in quality attributes. Poorly implemented software can decline in attributes such as maintainability and reusability. Software that is not maintainable has great chances of having a short period in the market.

In order to improve problem points in the software, it is possible to use refactoring techniques. Software refactoring consists of improvements in the internal structure of a program, through transformations, maintaining the original behavior of the program.

The Fowler catalog is one of the leading references in terms of refactoring. In it, refactorings are described informally, through determined steps, instructions for verifying the immutability of behavior and demonstrative examples. However, the use of such an approach to refactoring can cause errors and be very laborious, due to the manual way of doing it.

Aiming to save developers from having to manually refactor their code, development tools such as Eclipse, Netbeans and Intellij Idea automate refactorings similar to those in the Fowler catalog. However, for various reasons, some refactorings available in these tools do not guarantee the preservation of program behavior.

Studies show that it is possible to formalize refactorings so that program behavior can be guaranteed under certain conditions. One of these works is based on algebraic programming laws combined with laws for object-oriented programming. Using these laws, you can formalize any transformations between programs implemented in some object-oriented language, such as Java or C ++, and prove that the behavior is preserved.

Observing the need for development tools to offer reliable refactorings, this work proposes a strategy for implementing correct refactorings in the Eclipse environment. This approach is based on the formalization of refactoring, implementation of refactorings following the formulated conditions and handling eventual cases of aliasing with techniques of data flow analysis.

As a case study, we will deploy a secure version of Inline Temp refactoring for Java, performing patches and incrementing the version available in Eclipse. The static conditions that refactoring requires are implemented and verified normally. However, for the dynamic conditions, which involve verification of aliasing, an approach is proposed that generates evidentiary obligations described as assertive.

It is a known fact that a good part of the developers have little contact with formalisms and formal refactorings. In this way, the instrumentation of the code with assertions can be little pragmatic. Aiming to reduce the test obligations generated by dynamic analysis, this work performs a study on data flow analysis, techniques and tools that implement pointer analysis. Thus, it will be possible not to implement a certain piece of code with assertions, when it does not show signs of aliasing.

Keywords: Refactorings, Eclipse, Aliasing, Verification, Alias Analysis.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

AST	Abstract Syntax Tree
IBM	Internacional Business Machines
JDT	Java Development Toolkit
PDE	Plugin Development Environment
RCP	Rich Client Platform
SWT	Standard Widget Toolkit
WALA	Watson Libraries for Analysis

Lista de símbolos

Γ	Letra grega Gama
Λ	Lambda
ζ	Letra grega minúscula zeta
$\in$	Pertence

Sumário

1 Introdução

O código-fonte é um dos elementos primordiais na composição de um software. No decorrer de um projeto de desenvolvimento de software, o código passa por modificações visando melhorias em atributos de qualidade, como desempenho, legibilidade ou modularidade, adequações para extensões ou alterações de requisitos. Caso as alterações não sejam bem implementadas, o software pode decair em critérios de qualidade, podendo até resultar no fim do seu ciclo de vida (PRESSMAN, 2009). A fim de evitar estas degradações no código, projetistas e desenvolvedores podem recorrer ao uso de refatorações.

Uma refatoração consiste de uma ou mais alterações no código visando a melhoria da estrutura interna do programa e, ao mesmo tempo, garantindo a manutenção do comportamento do programa após a operação (FOWLER et al., 1999).

Para guiar desenvolvedores a se apropriar do uso de refatorações, foi proposto um catálogo em (FOWLER et al., 1999), onde é descrito um conjunto significativo de refatorações. Desde então, o catálogo tem sido ampliado e atualizado em páginas da *web*[1]. Contudo, as refatorações são prescritas de forma a serem feitas manualmente, podendo causar erros e necessitando de muita atenção do desenvolvedor.

De modo a automatizar o processo de refatoração, pode-se utilizar uma das inúmeras ferramentas existentes atualmente, como Netbeans, Eclipse ou Intellij Idea. Tais ambientes implementam refatorações propostas por Fowler et al. (1999) e algumas variantes. Contudo, nenhuma destas ferramentas abrange todo o catálogo, assim como não garantem a correção das refatorações disponíveis.

Em particular, o Eclipse[2] fornece suporte a refatorações desde a versão 1.0 e desde então vem incrementando o seu catálogo, assim como fortalecendo a confiabilidade sobre as refatorações implementadas (FUHRER; KELLER; KIEīUN, 2007). Porém, em determinados

[1] https://refactoring.com
[2] https://eclipse.org

casos, o Eclipse não consegue garantir que o comportamento dos programas refatorados sejam idênticos aos originais.

Observando que nem todas as refatorações oferecidas pelas ferramentas preservam o comportamento do programa, em (LUCERO, 2015) foi proposto que refatorações fossem formalizadas e provadas corretas. Para isso, utilizou-se um conjunto de leis algébricas que definem transformações de programas orientados a objetos (HOARE et al., 1987; BORBA et al., 2004; CORNÉLIO; CAVALCANTI; SAMPAIO, 2010).

A fim de preencher as lacunas que ferramentas como o Eclipse deixam em relação a condições para que determinadas refatorações preservem comportamento, em (SANTOS; LUCERO, 2017) propõe-se que as implementações se baseiem em refatorações formais como as de Lucero (2015). De fato, em (SANTOS; LUCERO, 2017) foi corroborado que a refatoração *Inline Temp* do Eclipse não é confiável quanto à preservação do comportamento e então foi proposta uma implementação guiada na formalização dada no trabalho de Lucero (2015).

Para que a refatoração seja considerada segura, é necessário que as condições formalizadas em (LUCERO, 2015) sejam verificadas. Entretanto, somente condições estáticas podem ser checadas antes da execução do código. Em geral, condições dinâmicas que dependam do retorno de métodos ou que contenham operações com ponteiros e possam gerar *aliasing* não podem ser totalmente avaliadas por uma ferramenta. Santos e Lucero (2017) propõem que condições dinâmicas sejam abordadas através de obrigações de prova estabelecidas com assertivas dentro do código.

De um ponto de vista teórico, gerar obrigações de prova como assertivas para garantir a solidez da refatoração é uma abordagem interessante, mas não é pragmática. Desenvolvedores de software dificilmente têm familiaridade com formalismos, ferramentas para desenvolvimento formal e nem com provadores de teoremas.

Este trabalho propõe uma abordagem para o desenvolvimento de refatorações seguras para o Eclipse, aplicando o estudo sobre refatorações formais. Como estudo de caso, implementaremos a refatoração *Inline Temp* na ferramenta Eclipse.

Propõe-se também o estudo de técnicas de análise de fluxo de dados para minimizar a geração de obrigações de prova. Especificamente, propõe-se a complementação da refatoração segura *Inline Temp*, proposta em (SANTOS; LUCERO, 2017), destacando como a análise de ponteiros pode ser utilizada para eliminar algumas das obrigações de prova. Em especial, esta proposta é baseada na ferramenta WALA, a qual tem se mostrado bastante eficaz (BODDEN, 2012; ISHIZAKI; DAIJAVAD; NAKATANI, 2011; MADHAVAN; KOMONDOOR, 2011).

Este trabalho está organizado em 5 capítulos, sendo o capítulo 1 responsável pela introdução ao conteúdo abordado.

O capítulo 2 apresenta um estudo inicial sobre refatorações, buscando apresentar a problemática em realizar refatorações de forma manual. Alguns dos principais catálogos de

refatorações também serão apresentados nesta seção. Exploraremos também, as ferramentas que disponibilizam a utilização de refatorações de forma automática. Em especial, trabalharemos mais a fundo com a ferramenta Eclipse, que será a utilizada no estudo de caso. Para que não haja lacunas de conhecimento nas seções posteriores, apresentaremos o *framework* de refatorações do Eclipse. Tal *framework* é fundamental no desenvolvimento de refatorações para a ferramenta.

No capítulo 3, apresentamos a abordagem utilizada para desenvolver refatorações seguras, utilizando-se do conhecimento de refatorações e leis algébricas.

No capítulo 4, estudaremos análise de fluxo de dados. As técnicas de análise de fluxo de dados são largamente utilizadas para, por exemplo, a descoberta de *aliasing* em um programa. Utilizaremos conceitos e técnicas de análise de fluxo de dados para propor uma abordagem inicial para a redução das obrigações de prova.

No capítulo 5 apresentamos as conclusões acerca do trabalho e as contribuições geradas. Alguns trabalhos futuros também serão propostos, levando em consideração o conteúdo abordado neste trabalho.

2
Refatoração

Neste capítulo, abordaremos técnicas de refatoração, definindo, exemplificando e explicando como o processo é definido em um catálogo de refatorações e como ocorre em ferramentas automáticas, como o Eclipse.

Refatorações são transformações no código que tem como objetivo a melhoria da estrutura interna, de modo que seja mais fácil de ser compreendido e que modificações futuras não sejam custosas. Estas transformações só são consideradas como refatorações, caso o comportamento do programa seja preservado (FOWLER et al., 1999). Um conjunto de refatorações define um catálogo, e um dos catálogos de refatorações mais conhecidos é o de Fowler et al. (1999).

2.1 Catálogo de Refatorações de Fowler

No catálogo apresentado em (FOWLER et al., 1999), as refatorações são descritas informalmente e para garantir a equivalência dos programas, depois da invocação de uma ou mais refatorações, são recomendados pequenos passos de transformação, que incluem verificações do compilador e execução de testes. O esquema utilizado para apresentação e explicação das refatorações segue os passos:

1. Exemplificação abstrata para o uso da refatoração.
2. Motivação necessária para o uso da refatoração.
3. Mecânica da refatoração.
4. Discussão de exemplos concretos.

A fim de exemplificar o esquema, analisemos a refatoração *Extract Method* aplicada ao código 1, obtendo como resultado o código 2.

Código 1 – Código antes da refatoração

```
public void soma(int fst, int snd){
   int result = fst + snd;
   System.out.println(result);
}
```

(→)

Código 2 – Código após refatoração

```
public void soma(int fst, int snd){
   int result = fst + snd;
   print(result);
}
private void print(int result) {
   System.out.println(result);
}
```

A refatoração *Extract Method* deve ser executada quando se vê um método que seja muito longo ou um código que precise de um comentário para ter seu propósito compreendido. Assim, este fragmento de código pode ser transformado em um novo método. Os benefícios na utilização desta refatoração são: maiores chances de um método ser usado por outros quando possui granularidade baixa, métodos de maior nível são lidos como se fossem uma série de comentários e sobrecarga mais fácil quando os métodos possuem granularidade maior. A mecânica da refatoração pode ser descrita nos seguintes tópicos:

- Crie um novo método e dê a ele um nome que o caracterize.
  - Se o código que quiser extrair for demasiado simples, você deve extraí-lo se o nome do novo método revelar a intenção do código de uma maneira melhor.
- Copie o código extraído do método de origem para o novo método.
- Busque no código extraído referências a variáveis que sejam locais no escopo do método fonte. Essas variáveis são os parâmetros e variáveis locais no método de origem.
- Veja se há variáveis locais usadas apenas dentro do código extraído. Caso haja, declare-as dentro do novo método como variáveis temporárias.
- Veja se algumas dessas variáveis de escopo local são modificadas pelo código extraído. Caso alguma seja, verifique se você pode tratar o código extraído como uma consulta e atribuir o resultado da consulta à variável em questão. Se isso for difícil, ou se houver mais de uma variável nessa situação, você não pode extrair o método dessa forma.
- Passe para o novo método como parâmetros as variáveis de escopo local que sejam lidas pelo código extraído.
- Compile quando você tiver tratado todas as variáveis locais.
- Substitua o código extraído no método fonte por uma chamada ao novo método.

  - Se você tiver movido qualquer variável temporária para o novo método, verifique se elas foram declaradas fora do código extraído. Se foram, você pode agora remover a declaração.

- Compile e teste.

Utilizando a mecânica apresentada, obtém-se certa confiabilidade de que não são introduzidos erros de compilação e que o comportamento é preservado. Contudo, a aplicação das transformações observada em (FOWLER et al., 1999) é realizada manualmente. Desta forma, o desenvolvedor pode cometer erros de entendimento e tornar o programa diferente do que havia implementado inicialmente. Para prevenir eventuais erros e livrar o desenvolvedor de realizar a refatoração passo a passo, ambientes de desenvolvimento têm se dedicado ao estudo e desenvolvimento de módulos que permitam que refatorações sejam realizadas de forma automática.

2.2 Refatorações Automáticas

Ferramentas de desenvolvimento como Eclipse, Netbeans e Intellij Idea, têm implementado seus próprios catálogos de refatorações, oferecendo ao desenvolvedor recursos para realizar automaticamente as respectivas transformações e verificação de condições (KIM; ZIMMERMANN; NAGAPPAN, 2012; PINTO; KAMEI, 2013). As condições verificadas evitam a introdução de erros de compilação e, em alguns casos, garantem a preservação do comportamento.

Em geral, as refatorações providas por estas ferramentas são variantes ou composições de refatorações descritas em (FOWLER et al., 1999). Um sumário da cobertura de cada uma destas ferramentas, tomando como referência o catálogo de Fowler et al. (1999), encontra-se no apêndice A. Como um exemplo vemos abaixo, na tabela 1, as refatorações relacionadas a simplificações de expressões condicionais.

As refatorações presentes na tabela 1 envolvem aspectos relacionados à simplificação de expressões condicionais, onde nota-se que, nenhum dos ambientes de desenvolvimento avaliados implementa as mesmas. Estas refatorações são pontos passíveis de exploração.

A ferramenta automática escolhida para ser utilizada neste projeto foi o Eclipse. Dentre as possibilidades disponíveis, foram avaliadas as ferramentas que fornecem suporte a linguagens orientadas a objetos, e, em particular, o Java. A escolha do Eclipse se deu pela documentação disponível, conhecimento acerca da ferramenta e por ser uma ferramenta de código-aberto.

Tabela 1 – Relação da refatorações relacionadas a simplificações de expressões condicionais.

Refatoração	Fowler	Eclipse	Netbeans	Intellij
Decompor Condicional	Sim	Não	Não	Não
Consolidar Expressão Condicional	Sim	Não	Não	Não
Consolidar Fragmentos Condicionais Duplicados	Sim	Não	Não	Não
Introduzir Asserção	Sim	Não	Não	Não
Introduzir Objeto Nulo	Sim	Não	Não	Não
Remover *Flag* de Controle	Sim	Não	Não	Não
Substituir Condição Aninhada por Cláusulas Guarda	Sim	Não	Não	Não
Substituir Comando Condicional por Polimorfismo	Sim	Não	Não	Não

2.3 Refatorações no Eclipse

O Eclipse é um um ambiente de desenvolvimento integrado que fornece ferramentas para desenvolvimento em diversas linguagens, como Java, C/C++ e Php. É uma ferramenta extensível, aberta e portável, e através de *plugins*, diversos serviços e linguagens podem ser combinados, criando um ambiente amplo e integrável.

Os principais componentes são: RCP (*Rich Client Platform*), a plataforma de desenvolvimento; JDT (*Java Development Tools*), um conjunto de *plugins* que adicionam à plataforma Eclipse uma IDE (*Integrated Development Environment*) Java completa; PDE (*Plug-in Development Environment*), oferece ferramentas para criar, desenvolver, testar, depurar, criar e implantar *plugins*, fragmentos, recursos, sites de atualização e produtos RCP do Eclipse. A arquitetura está disposta conforme a figura 1.

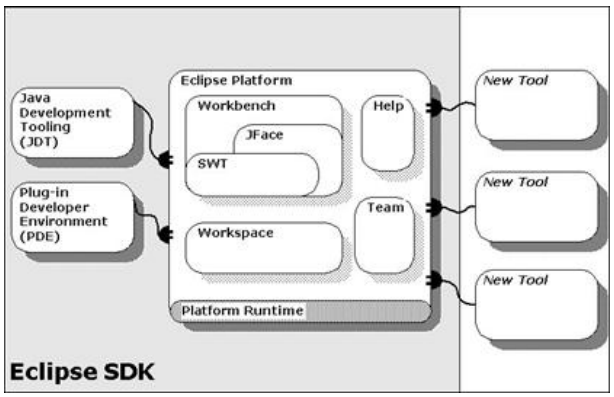

Figura 1 – Arquitetura do Eclipse.

Fonte: (ECLIPSE, 2017)

Na figura 1, podemos ver que na estrutura interna da plataforma de execução, estão contidos o workbench (ambiente de desenvolvimento de projetos), o SWT (*Standard Widget Toolkit*)

e JFace, que fornecem suporte ao desenvolvimento de interfaces; o workspace (diretório que abriga os projetos criados) e os componentes Help (ajuda e documentação) e Team (responsável por fornecer integração de ferramentas de repositório no Eclipse). Nota-se que os componentes JDT e PDE são acoplados à plataforma de execução, ou seja, apenas utilizam seus recursos para fornecer suporte ao desenvolvimento de aplicações. Novos *plugins* e ferramentas que são desenvolvidos com JDT e PDE também podem se utilizar da plataforma de execução.

O Eclipse fornece um extenso suporte[1] para refatorações e segue um *framework* que padroniza a forma com que as mesmas são codificadas.

O *framework* de refatorações do Eclipse propõe que cada refatoração seja implementada como uma subclasse de `Refactoring`. A classe abstrata `Refactoring` possui dois tipos de métodos: os métodos que são utilizados para checar se as transformações não implicarão na introdução de erros de compilação ou na mudança de comportamento do programa (`checkInitialConditions()` e `checkFinalConditions()`) e os métodos que executam modificações no projeto (`createChange()`). O código 3 dá uma visão simplificada da classe `Refactoring`.

Código 3 – Representação da classe Refactoring

```
abstract class Refactoring extends PlatformObject{
   private Object fValidationContext;
   public final void setValidationContext(Object context);
   public final Object getValidationContext();
   public abstract RefactoringStatus checkInitialConditions(IProgressMonitor
       pm);
   public abstract RefactoringStatus checkFinalConditions(IProgressMonitor pm);
   public abstract Change createChange(IProgressMonitor pm);
}
```

A execução de uma refatoração pelo *framework* de refatorações do Eclipse é realizada a partir da utilização dos métodos sobrescritos da classe abstrata `Refactoring`, apresentada no código 3. Os métodos `setValidationContext()` e `getValidationContext()` dizem respeito ao contexto ao qual a refatoração está sendo realizada.

Ao iniciarmos o processo de refatoração, é obtida uma representação do programa em formato de uma árvore sintática abstrata, em inglês *Abstract Syntax Tree* (AST). A AST é avaliada, para verificar se há a possibilidade de prosseguir com a refatoração no método `checkInitialConditions()`. Em seguida, o usuário fornece os dados adicionais necessários e ocorre uma nova verificação a fim de reavaliar a validade da transformação sobre o código; isso ocorre no método `checkFinalConditions()`. Com todas as condições verificadas e caso não haja falha em nenhum dos passos anteriores, o Eclipse faz a transformação da AST antiga para

[1] http://help.eclipse.org

uma AST modificada e reescreve o código com o método `createChange()`. A figura 2 mostra a sistematização desse processo.

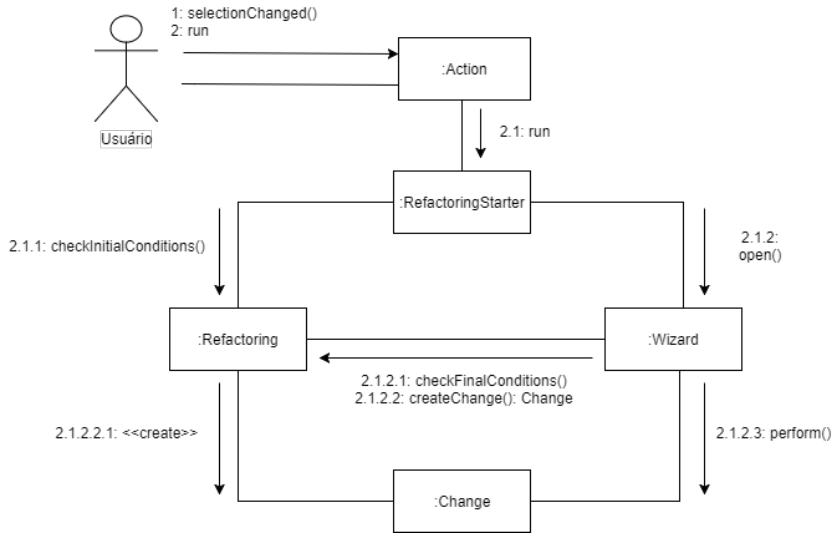

Figura 2 – Representação de uma refatoração em diagrama de comunicação.

Adaptado de (ECLIPSE, 2017).

Quando o usuário realiza uma seleção de texto e aciona a refatoração desejada, a ação associada à refatoração, que será uma subclasse de `Action`, ativará o método `run()` e dará início à refatoração, criando uma instância da classe `RefactoringStarter`. A partir daí, duas ações ocorrem: o método `checkInitialConditions()` da classe `Refactoring` é chamado, a fim de avaliar se a refatoração pode ser realizada no código selecionado e o método `open()` da classe `Wizard` é chamado para que o usuário insira os dados adicionais, necessários à refatoração, quando houver. Quando esses dados forem fornecidos a `Wizard`, o método `checkFinalConditions()` da classe `Refactoring` verifica se a refatoração pode ser aplicada com os novos dados e, se a resposta for sim, o método `createChange()` cria uma nova AST para o código refatorado. Então, cria-se uma instância de `Change` e o `Wizard` chama o método `perform()` de `Change`, que sobrescreve o código original com o refatorado.

No próximo capítulo, utilizaremos os conceitos estudados sobre refatorações e sobre o *framework* de refatorações do Eclipse para propor refatorações seguras, implementadas no Eclipse, tendo como estudo de caso a refatoração *Inline Temp*.

3

Refatorações Seguras

Como mencionado no capítulo anterior, o processo de refatoração é simplificado ao utilizar ferramentas automáticas, reduzindo a complexidade das transformações e a possibilidade de erros pelo desenvolvedor. Contudo, nem sempre refatorações implementadas nestas ferramentas são seguras, pois, existem algumas que modificam o comportamento do programa. Um exemplo é a refatoração *Inline Temp* provida pelo Eclipse, como ilustrado pelos códigos 4 e 5. É fácil de perceber que o método `operation` do código 5 não retorna o mesmo valor que o método `operation` do código 4.

Código 4 – Programa original.

```
public int operation(int ext){
   int temp = ext*ext;
   ext = ext + 1;
   return temp;
}
```

(→)

Código 5 – Programa gerado pelo Eclipse, após refatoração

```
public int operation(int ext){
   ext = ext + 1;
   return ext*ext;
}
```

De modo a tornar as refatorações seguras, em (LUCERO, 2015; BORBA et al., 2004; CORNÉLIO; CAVALCANTI; SAMPAIO, 2010; LUCERO; NAUMANN; SAMPAIO, 2013) propõe-se uma abordagem que se utiliza de leis de programação para formalizar e provar refatorações. Em (SANTOS; LUCERO, 2017), propõe-se que estes trabalhos sejam a base para propor correções a refatorações implementadas na ferramenta Eclipse e, como um estudo de caso, apresenta-se uma implementação segura para a refatoração *Inline Temp*.

Neste capítulo, abordaremos leis algébricas de programação e uma formalização de refatorações que se utiliza destas leis. Como estudo de caso, mostraremos como tornar segura a refatoração do Eclipse *Inline Temp*. Na seção 3.1 serão apresentadas as leis algébricas de programação e, na seção 3.2, a proposta de refatorações seguras fundamentadas nas leis. Na

seção 3.3, apresenta-se a abordagem para tornar refatorações do Eclipse corretas, verificando condições estáticas e instrumentando os programas com assertivas que garantem condições dinâmicas.

3.1 Leis de Programação

As leis de programação, formuladas inicialmente por Hoare et al. (1987), estabelecem uma fundamentação algébrica para a programação. Nesse trabalho, contempla-se apenas uma linguagem reduzida cuja sintaxe contempla as seguintes construções.

$$
\begin{array}{rcll}
c & ::= & \overline{x} := \overline{e} \mid \bot \mid \textbf{skip} & \text{atribuição simultânea, } \textit{abort}, \textit{skip} \\
 & & \mid c\,;c \mid c \lhd e \rhd c \mid c \cup c & \text{sequência, condicional, não-determinismo} \\
 & & \mid \mu X @ c \mid X & \text{recursão, chamada recursiva}
\end{array}
$$

Na atribuição $\overline{x} := \overline{e}$, o lado esquerdo $\overline{x}$ representa uma lista de variáveis distintas e o lado direito $\overline{E}$ uma lista de expressões, sendo que as duas listas devem ser do mesmo tamanho. O comando $\bot$, denominado *abort*, representa o comportamento de falha de um programa. Sua execução é não determinística podendo realizar qualquer alteração no estado do programa ou até divergir em um *loop* infinito. Por outro lado, **skip** representa o comando que não faz nada, ou seja, deixa tudo de forma inalterada. O sequenciamento de dois comandos é expresso com o operador infixo ";". O comando condicional $c_1 \lhd b \rhd c_2$ atua como o comando $\textbf{if}\,(b)\,c_1\,\textbf{else}\,c_2$ do Java. O comando não-determinístico $c_1 \cup c_2$ realiza uma escolha arbitrária entre c_1 e c_2. A notação $\mu X @ c$ representa um comando recursivo no qual cada ocorrência de X dentro de c é uma chamada recursiva.

Propriedades gerais dos programas são expressas através de leis expressas como equações que relacionam elementos da linguagem de programação. As variáveis destas equações representam entidades tais como comandos e expressões. Por exemplo, a seguinte lei vale para qualquer comando c.

$$c;\textbf{skip} = \textbf{skip};c = c \tag{3.1}$$

A lei expressa que se executamos c seguido de **skip** ou **skip** seguido de c, ambos casos tem o mesmo efeito de executar somente c. Na terminologia usada em álgebra podemos dizer que o comando **skip** é a unidade para a operação de sequenciamento ";".

De fato, um conjunto significativo de leis expressam propriedades bastantes familiares na álgebra. Para mencionar algumas temos que: *abort* é um elemento absortivo para o sequenciamento, o sequenciamento é associativo, o não determinismo é comutativo e associativo, e o sequenciamento distribui sobre o não determinismo. No entanto, algumas leis são bastante peculiares à programação. Por exemplo, a seguinte lei sobre a atribuição estabelece que duas

atribuições consecutivas sobre as mesmas variáveis podem ser transformadas em uma única atribuição.

$$\overline{x} := \overline{e};\ \overline{x} := \overline{d} \ = \ \overline{x} := \overline{d}^{\overline{x}}_{\overline{e}} \lhd \mathfrak{D} \overline{e} \rhd \bot \tag{3.2}$$

A notação $d^{\overline{x}}_{\overline{e}}$ representa a substituição simultânea feita sobre d de tal forma que cada uma das variáveis em $\overline{x}$ é substituída pela correspondente expressão em $\overline{e}$. Assim, a grosso modo, a lei diz que se atribuímos a $\overline{x}$ os valores de $\overline{e}$ e, imediatamente depois, atribuímos novamente para $\overline{x}$ os valores de $\overline{d}$ então podemos obter o mesmo resultado realizando somente uma única atribuição para $\overline{x}$ dos valores de $d^{\overline{x}}_{\overline{e}}$. Porém, já que a atribuição $\overline{x} := \overline{e}$ no lado esquerdo da leis sempre avalia $\overline{e}$, no lado direito exigimos que $\overline{e}$ esteja definido. A formulação de todas as leis, assim como explicações acerca delas, encontram-se em (HOARE et al., 1987).

3.2 Refatorações Formais

No trabalho de Borba et al. (2004) as leis da programação são estendidas para considerar uma linguagem orientada a objetos. Tais leis, denominadas leis de classes, podem ser utilizadas como fundamentação para transformações de programas orientados a objetos.

Para exemplificar como são postuladas as leis de classes, mostramos a baixo uma das leis presentes em (BORBA et al., 2004), a *move redefined method to superclass*. Esta lei destaca o comportamento do mecanismo de ligação dinâmica na chamada a métodos. Ela estabelece que podemos unir uma declaração de método e sua redefinição em uma única declaração na superclasse. O método resultante escolhe o comportamento apropriado através de teste de tipos.

```
class B extends A
  ads
  meth m ≙ ( sig@b )
  mts
end
class C extends B
  ads′
  meth m ≙ ( sig@b′ )
  mts′
end
```

$=_{cds,c}$

```
class B extends A
  ads
  meth m ≙ ( sig@
   b ◁not(self is C)▷ b′
   )
  mts
end
class C extends B
  ads′
  mts′
end
```

cláusulas

($\leftrightarrow$) (1) campos **super** e privados não aparecem em b'; e

(2) **super**.m não aparece em mts'.

($\rightarrow$) b' não contém uncast ocorrências de **self** nem expressões contendo a forma $((C)\textbf{self}).a$ para cada campo privado a em ads';

$(\leftarrow)$ m não é declarado em mts'.

As restrições apresentadas na lei acima denotam a direção na qual a transformação está sendo realizada. Mais à frente, veremos que isso influencia na análise de uma refatoração formal.

O trabalho de Borba et al. (2004) foi o primeiro trabalho que usou a abordagem algébrica para formalizar e provar refatorações tendo como fundamento leis de programação. No entanto, como a linguagem de programação não tem referências, um conceito fundamental em orientação a objetos, o seu escopo é limitado. A complementação do trabalho de Borba et al. (2004), permitindo o estudo de refatorações que envolvem programas que usam referências é realizada em (LUCERO, 2015). Neste último trabalho, refatorações são descritas como equações entre programas junto com condições que os programas devem satisfazer para que a equação seja válida. Algumas destas condições podem ser verificadas estaticamente, sem precisar executar os programas. No entanto outras são de natureza dinâmica.

A fim de exemplificação, analisemos a formalização da refatoração *Inline Temp* contida na proposta de Lucero (2015):

$$\textbf{var}\ t : T \bullet t := e;\ c[t] = [\textbf{isdef}(e)];\ c[e]$$

Onde $e : T$.

Cláusulas

$(\rightarrow)$ (1) t é somente-leitura em $c[t]$; (2) o valor de e não muda ao longo de $c[t]$.

$(\leftarrow)$ (1) t é nova em $c[e]$; (2) o valor de e não muda ao longo de $c[t]$.

A notação $\textbf{var}\ t : T \bullet c$ define um bloco de comandos que declara a variável local t e cujo corpo é o comando local c. A expressão $[\textbf{isdef}(e)]$ é uma assertiva (*assertion*) afirmando que não haverá erro na avaliação de e, ao passo que $c[e]$ é o comando obtido a partir de $c[t]$ substituindo toda ocorrência de t por e. A prova desta refatoração é dada em (LUCERO, 2015) mediante a aplicação das leis de programação orientada a objetos com referências.

Quando aplicada no sentido $(\rightarrow)$, a refatoração apresentada se corresponde com *Inline Temp*, já quando realizada no sentido $(\leftarrow)$, a refatoração corresponde-se à *Extract Temp*, ambas do catálogo de Fowler et al. (1999). Observe que esta descrição formal da refatoração requer duas checagens quando aplicada no sentido $(\rightarrow)$: uma estática e outra dinâmica. A verificação de que uma variável t é somente-leitura sempre pode ser feita estaticamente. Já a verificação de que e não muda ao longo da execução de $c[t]$, que significa que o valor da expressão e não é modificado durante a execução de $c[t]$, só pode ser verificado dinamicamente.

Na próxima seção, veremos a proposta de Santos e Lucero (2017) para lidar com condições dinâmicas através da geração de obrigações de prova dentro do comando $c[t]$, explicitadas como assertivas que garantem a invariabilidade de e.

3.3 Estratégia para Implementar Refatorações Seguras

Como visto no exemplo inicial deste capítulo (códigos 4 e 5), a refatoração *Inline Temp* do Eclipse não garante que o comportamento do programa original seja preservado. Essa inconsistência se deve à falta de verificação de condições sobre as quais a refatoração é correta. Tais condições são especificadas nas refatorações formais propostas em (LUCERO, 2015). Assim, adotando a formalização da refatoração *Inline Temp*, determinada na subseção anterior, as seguintes condições se tornam necessárias:

($\rightarrow$) (1) t é somente-leitura em $c[t]$; (2) o valor de e não muda ao longo de $c[t]$.

Em relação à condição (1), onde exige-se que a variável temporária t seja somente-leitura no contexto $c[t]$, verifica-se que o Eclipse implementa corretamente esta checagem estática. Contudo, o Eclipse não realiza a checagem da condição (2), onde se requer que o valor da expressão e não seja modificado em $c[t]$. Neste trabalho assume-se como simplificação que a expressão e não contém chamadas a métodos. Desta forma, para satisfazer a condição (2) podemos focar somente nas variáveis e campos referenciadas dentro da expressão e para que não sejam alteradas. Contudo, devido à possibilidade de *aliasing*, isto pode exigir verificações dinâmicas.

Santos e Lucero (2017) apresentam um método para lidar com condições dinâmicas como a (2) da *Inline Method* exemplificando com uma proposta de implementação desta refatoração para o Eclipse. Considerando a simplificação de que a expressão e não contém chamadas a métodos, são propostas condições mais fortes, porém seguras e que, na prática, não comprometem a aplicabilidade da refatoração.

Em resumo, requer-se que:

(a) variáveis envolvidas em e sejam somente de leitura, o que pode ser verificado estaticamente.

(b) campos envolvidos em e não sejam alterados em $c[t]$.

Para satisfazer a condição b), geramos obrigações de prova instrumentadas como assertivas que garantem a preservação do valor inicial da expressão e ao longo do comando $c[t]$. As assertivas são inseridas imediatamente antes de atribuições que podem potencialmente alterar e e, imediatamente depois de chamadas a métodos, pois potencialmente podem haver alterações via *aliasing*. O exemplo descrito a seguir, nos códigos 3.3 e 3.3, é útil para ilustrar a estrategia adotada na implementação. Neste exemplo a variável foco da refatoração é `t` e o resultado expande cada ocorrência de `t` por `x.f+k`.

```
public SomeType method(){
   int t = x.f + k;
   ... t ...
   e.f = exp;
   ... t ...
}
```

(→)

```
public SomeType method(){
   ... x.f + k ...
   assert x != e;
   e.f = exp;
   ... x.f + k ...
}
```

```
public SomeType method(){
   int t = x.f + k;
   ... t ...
   d.m();
   ... t ...
}
```

(→)

```
public SomeType method(){
   final int tf = x.f;
   ... x.f + k ...
   d.m();
   assert x.f == tf;
   ... x.f + k ...
}
```

A instrumentação com assertivas é uma forma de garantir que o *aliasing* não irá ocorrer, desde que o desenvolvedor garanta as obrigações de prova. No exemplo acima, as assertivas são utilizadas para garantir que o campo `x.f` não seja modificado nem via *aliasing* em atribuições, nem através de chamadas a métodos.

Assim, se o código original contém uma atribuição com lado esquerdo $e.f$, precisamos garantir que $e.f$ não está em *aliasing* com $x.f$, ou seja, que e é diferente de x.

Outra verificação dinâmica é necessária quando há chamadas a métodos, pois eles podem alterar valores de campos. Para isto, colocamos como obrigação de prova para o desenvolvedor, garantir que o valor dos campos utilizados na expressão não mudam após a chamada de métodos. A variável `tf`, que tecnicamente é uma variável fantasma (*ghost*), é utilizada para guardar o valor inicial do campo `x.f` e a assertiva após a chamada `d.m()` garante que o valor `x.f` foi inalterado pelo método `m`.

Para sermos exatos, com relação à condição (→)(2) da refatoração formal, nossa implementação faz uma exigência mais forte da realmente demandada, a qual somente requer que o valor de `x.f + k` não seja alterado durante a execução do corpo de `m`, no entanto nada diz sobre `x`, `k` ou `x.f`. Porém, analisando usos práticos desta refatoração, pode-se confirmar que isto não compromete a aplicabilidade da refatoração.

Na próxima seção, mostraremos como implementamos esta abordagem no ambiente Eclipse, resultando em uma refatoração *Inline Temp* segura.

3.4 Implementação da Refatoração *Inline Temp*

No contexto geral, a implementação da refatoração se dá a partir de uma classe filha da classe `InlineTempRefactoring` do Eclipse, denominada `SoundInlineTempRefactoring`.

Nela, iremos realizar a implementação das condições apresentadas na seção anterior, de modo que uma versão segura da *Inline Temp* seja proporcionada.

Com o intuito de implementar a verificação da condição estática relativa à refatoração *Inline Temp*, apresentada anteriormente, iremos percorrer a árvore sintática abstrata que o Eclipse utiliza para representar os programas utilizando o padrão de projeto *visitor*. Partindo do nó correspondente ao método que contém a expressão atribuída à variável, visita-se a árvore em ordem verificando se alguma das variáveis utilizadas na expressão é modificada. Caso isso ocorra, a refatoração não pode ser realizada. O pseudocódigo simplificado relativo a essa verificação é apresentado no código 6.

Código 6 – Representação da checagem de condições estáticas

```
public boolean checkReadOnly(CompilationUnit un) {
      boolean error = false;
      right = fVariableDeclaration.getInitializer();
      AssignmentVisitor assignmentVisitor = new AssignmentVisitor();
      SimpleNameVisitor simpleNameVisitor = new SimpleNameVisitor();
      QualifiedNameVisitor qualifiedNameVisitor = new QualifiedNameVisitor();
      right.accept(simpleNameVisitor);
      List<SimpleName> variables = simpleNameVisitor.getVariables();
      right.accept(qualifiedNameVisitor);
      List<QualifiedName> fieldList = qualifiedNameVisitor.getFields();
      MethodDeclaration methodVerified = getMethodDeclaration(un);
      methodVerified.accept(assignmentVisitor);
      List<String> elements = new ArrayList<String>();
      int expressionPosition = right.getStartPosition();
      for(Assignment assignment : assignmentVisitor.getAssignments()){
         Expression aux = assignment.getLeftHandSide();
         int assignPosition = aux.getStartPosition();
         if(assignPosition > expressionPosition)
            elements.add(convertToString(aux));
      }
      for(SimpleName sn : variables){
         if(elements.contains(convertToString(sn)))
            error = true;
      }
      for(QualifiedName qn : fieldList){
         if(elements.contains(convertToString(qn)))
            error = true;
      }
      return error;
   }
```

O primeiro passo da verificação acima consiste em extrair as variáveis contidas na expressão, utilizando o `simpleNameVisitor`, que faz uma visita nas variáveis existentes, e colocando-as em uma lista (`variables`). Isto é feito com o campo `fVariableDeclaration`, que se trata do nó de declaração de variável, ao qual a refatoração está sendo realizada. O

mesmo processo é feito para campos existentes na expressão com `qualifiedNameVisitor`, resultando na lista de campos `fieldList`. O segundo passo consiste em verificar as atribuições contidas no método, utilizando `assignmentVisitor`, que faz a visita em atribuições, e colocar as expressões do lado esquerdo na lista `elements`, desde que as atribuições estejam após à declaração da variável temporária no código. O último passo é verificar se as variáveis que estão na lista `elements` estão também na lista `variables` ou na lista `fieldList`, caso isso ocorra, o método `checkReadOnly()` deve retornar um *status* de erro.

Caso haja um ou mais campos na expressão, teremos que lidar com a possibilidade de *aliasing* acerca dos ponteiros. O código 7 lida com possíveis modificações do valor do campo entre ponteiros, dentro do método que contém a variável temporária. Em particular, o código 7 lida com atribuições feitas a campos no mesmo escopo onde a variável temporária será substituída.

Para garantir que não haja *aliasing*, utilizamos a instrumentação com assertivas explanada anteriormente. O método `doAssignmentAssertions()`, apresentado no código 7 se utiliza do passeio na árvore sintática abstrata, por meio do padrão de projeto *visitor*, a fim de encontrar campos contidos na expressão atribuída à variável temporária, representada por `right`, salvando-os em uma lista `fields`.

Caso exista um ou mais campos, verificamos com `assignmentVisitor` quais as atribuições existentes no método, representado pelo seu nó na árvore (`methodNode`). Se o lado esquerdo de alguma destas atribuições for um campo, que tenha o nome igual a algum dos campos em `fields`, instrumentamos o código com uma assertiva para verificar se o campo do lado esquerdo da atribuição possa estar em *aliasing* com algum campo em `fields`. As assertivas são inclusas na árvore abstrata original, após cada atribuição. Os métodos `recordModifications` e `rewrite` servem para efetivar a reescrita da árvore no código.

Similarmente, devemos verificar se chamadas a métodos, dentro do escopo em que a refatoração ocorre, não irão modificar o valor de campos contidos na expressão. O código 8 apresenta a resolução.

Primeiramente, verificamos se existem campos na expressão e se se existem chamadas a métodos no corpo do método avaliado. Isso ocorre por meio de `qualifiedName` para campos e `methodInvocation` para as chamadas a métodos. As ocorrências de ambos são computados em duas listas, `fields` e `methods` respectivamente. Caso existam, para cada campo, é criada uma variável fantasma (*ghost*) para salvar o valor inicial do campo. Essa nova variável, representada por `vds` é inclusa na árvore antes da declaração da variável temporária. Após cada chamada a método, uma assertiva é criada para verificar se os campos não tiveram seus valores alterados. Esta verificação é feita com as variáveis fantasmas criadas anteriormente. Após todas as verificações, o código é atualizado com `un.rewrite()`.

Desta forma, finalizamos as condições (estáticas e dinâmicas) restantes que foram

Código 7 – Instrumentação de casos onde hajam campos na expressão

```
RefactoringStatus doAssignmentAssertions(CompilationUnit un, ASTNode
    methodNode) {
   Expression right = fVariableDeclaration.getInitializer();
   QualifiedNameVisitor qualifiedNameVisitor = new QualifiedNameVisitor();
   AssignmentVisitor assignmentVisitor = new AssignmentVisitor();
   right.accept(qualifiedNameVisitor);
   List<QualifiedName> fields = qualifiedNameVisitor.getFields();
   if(fields.length() > 0){
      AST ast = un.getAST();
      methodNode.accept(assignmentVisitor);
      un.recordModifications();
      for(Assignment assignment : assignmentVisitor.getAssignments()){
         Expression exp = assignment.getLeftHandSide();
         if(exp.getNodeType() == ASTNode.QUALIFIED_NAME){
            Name name = exp.getQualifier();
            SimpleName simpleName = exp.getName();
            for(QualifiedName qualified : fields){
               if(qualified.getName() == simpleName){
                  AssertionStatement assertStt = ast.newAssertStatement();
                  InfixExpression infixExp = ast.newInfixExpression();
                  infixExp.setOperator(InfixExpression.Operator.NOT_EQUALS);
                  infixExp.setLeftOperand(name);
                  infixExp.setRightOperand(qualified.getQualifier());
                  assertStt.setExpression(infixExp);
                  un.insertBefore(assertStt, assignment);
               }
            }
         }
      }
      un.rewrite();
   }
   return new RefactoringStatus();
}
```

propostas anteriormente. Com a implementação dessas condições, podemos garantir a solidez da refatoração *Inline Temp*, obtendo uma classe resultante, que se utiliza dos códigos 6, 7 e 8, e é apresentada no código 9.

A implementação apresentada anteriormente, assim como o código-fonte do *plugin* em desenvolvimento, está disponível no repositório oficial do projeto [1]. As classes principais deste trabalho estão no apêndice B.

No próximo capítulo, realizaremos um estudo sobre análise de fluxo de dados para investigar como o uso de ferramentas que implementam análise de ponteiros pode reduzir as obrigações de prova geradas nas implementações realizadas neste capítulo.

[1] http://github.com/igormoraisn/sound-refactorings

Código 8 – Representação de resolução de condições dinâmicas

```
RefactoringStatus doMethodAssertions(CompilationUnit un, ASTNode nodeSelected) {
   Expression right = fVariableDeclaration.getInitializer();
   QualifiedNameVisitor qualifiedName = new QualifiedNameVisitor();
   MethodInvocationVisitor methodInvocation = new MethodInvocationVisitor();
   un.recordModifications();
   right.accept(qualifiedName);
   List<QualifiedName> fields = qualifiedName.getFields();
   nodeSelected.accept();
   List<MethodInvocation> methods = methodInvocation.getMethods();
   if(fields.size() > 0 && methods.size() > 0){
      AST ast = un.getAST();
      int i=1;
      for(QualifiedName qualified : fields){
         VariableDeclarationFragment vd = ast.newVariableDeclarationFragment();
         Name name = ast.newName(convertToString(qualified.getQualifier()));
         SimpleName simpleName =
             ast.newSimpleName(convertToString(qualified.getName()));
         QualifiedName initializer = ast.newQualifiedName(name, simpleName);
         SimpleName varName = ast.newSimpleName("t"+ i);
         vd.setName(varName);
         vd.setInitializer(initializer);
         VariableDeclarationStatement vds =
             ast.newVariableDeclarationStatement(vd);
         listRewrite.insertBefore(vds, nodeSelected, null);
         i++;
      }
      for(MethodInvocation m : methods){
         i=1;
         for(QualifiedName qualified : fields){
            Name name = ast.newName(convertToString(qualified.getQualifier()));
            SimpleName simpleName =
                ast.newSimpleName(convertToString(qualified.getName()));
            QualifiedName left = ast.newQualifiedName(name, simpleName);
            SimpleName rightExp = ast.newSimpleName("t" + i);
            InfixExpression infixExpression = ast.newInfixExpression();
            infixExpression.setOperator(InfixExpression.Operator.EQUALS);
            infixExpression.setLeftOperand(left);
            infixExpression.setRightOperand(rightExp);
            AssertStatement as = ast.newAssertStatement();
            as.setExpression(infixExpression);
            listRewrite.insertAfter(as, m, null);
            i++;
         }
      }
   }
   un.rewrite();
   return new RefactoringStatus();
}
```

Código 9 – Método que realiza as checagens estáticas e instrumenta as checagens dinâmicas

```
public RefactoringStatus checkInitialConditions(IProgressMonitor pm){
   MethodDeclaration method = getMethodDeclaration;
   ASTParser parser = ASTParser.newParser(AST.JLS3);
   parser.setSource(fCu);
   CompilationUnit un = (CompilationUnit) parser.createAST(null);
   if(!checkReadOnly(un, method))
      return RefactoringStatus.createFatalError("One or most variables changed
          after expression");
   doAssignmentAssertions(un, method);
   doMethodAssertions(un, method);
   return new RefactoringStatus();
}
```

4

Análise de Ponteiros

A abordagem apresentada no capítulo 3 demonstra como projetar e implementar uma refatoração com o auxílio de formalizações. Porém, a estratégia utilizada requer condições fortes devido ao uso de assertivas para obter garantias de que não haja *aliasing*.

Uma forma de verificar se um determinado programa pode apresentar *aliasing* é realizando análise de ponteiros, uma técnica de análise de fluxo de dados. É importante alertar que esta técnica é conservadora, detectando apenas possibilidades de existência de *aliasing*. No entanto, ela é correta (*sound*) no sentido de que se não foi detectada a possibilidade de *aliasing*, de fato não haverá. Este capítulo apresentará informalmente a técnica de análise de fluxo de dados e algumas das suas aplicações.

4.1 Análise de Fluxo de Dados

Otimizações em compiladores realizam transformações no programa para melhorar sua eficiência, sem modificar seu comportamento. Determinadas otimizações de código dependem de análises que obtém informações do fluxo de dados de programas ao longo dos seus caminhos de execução (AHO et al., 2006).

Por exemplo, utilizando análise de fluxo de dados, é possível encontrar e eliminar subexpressões comuns, que requer determinar em quais pontos duas expressões idênticas textualmente resultam no mesmo valor ao longo de cada caminho possível dentro do programa (APPEL; PALSBERG, 2003). Outro exemplo ocorre quando um valor que não é usado depois é atribuído a uma variável. Assim, podemos eliminar esta atribuição, considerando-a como código morto.

Para realizar a análise do programa, é necessário uma representação do mesmo, de forma que se possa obter dados acerca dele, de forma sistemática. Em análise de fluxo de dados, a representação utilizada é um grafo de fluxo do programa. A figura 3 apresenta o grafo de fluxo de um determinado programa.

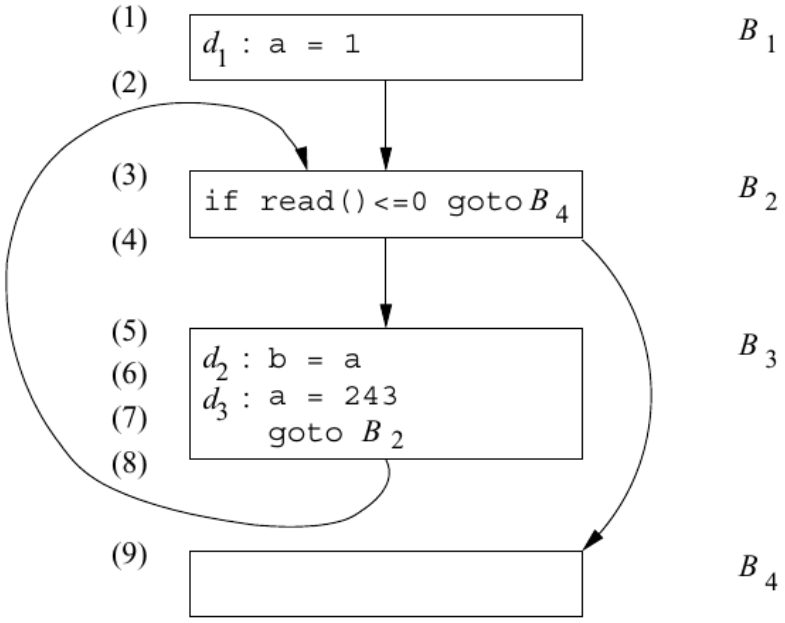

Figura 3 – Ilustração de um programa em um grafo de fluxo.

Fonte: (AHO et al., 2006)

Utilizando esta representação, é possível realizar diversos tipos de análises, como, por exemplo, análise de código morto ou expressões disponíveis.

Uma outra representação para um grafo de fluxo de dados é no formato de quádruplas. Normalmente, se utiliza o estado a $\leftarrow$ b $\oplus$ c com quatro componentes (a, b, c, $\oplus$). Esses estados são chamadas de quádruplas. Usamos $\oplus$ para representar uma operação binária arbitrária. Um compilador mais eficiente representaria quádruplas com seu próprio tipo de dados e traduziria de árvores para quádruplas tudo em uma única passagem.

Suponhamos que desejemos fazer eliminação de subexpressão comum, isto é, dado um programa que calcula x $\oplus$ y mais de uma vez, podemos eliminar um dos cálculos duplicados. Para encontrar lugares onde tais otimizações são possíveis, a noção de expressões disponíveis é útil.

Uma expressão x $\oplus$ y está disponível em um nó n no grafo de fluxo se, em cada caminho do nó de entrada do grafo para o nó n, x $\oplus$ y é calculado pelo menos uma vez e não há definições de x ou y desde a ocorrência mais recente de x $\oplus$ y nesse caminho. Podemos expressar isso em equações de fluxo de dados usando conjuntos de expressões *gen* e *kill*, para gerar e matar expressões, respectivamente, como explicado a seguir.

Qualquer nó que calcula $x \oplus y$ gera (*gen*) $x \oplus y$, e qualquer definição de x ou y mata (*kill*) $x \oplus y$. Um exemplo é apresentado na tabela 2.

Basicamente, $t \leftarrow b \oplus c$ gera a expressão $b \oplus c$. Mas $b \leftarrow b \oplus c$ faz com que não se gere $b \oplus c$, porque depois de $b \oplus c$ existe uma definição posterior de b. A declaração $gen[s] =$

Tabela 2 – *Gen* e *Kill* para expressões disponíveis.

Estado s	$gen(s)$	$kill(s)$
$t \leftarrow b \oplus c$	$\{b \oplus c\}$ - $kill[s]$	expressões contendo t
$t \leftarrow M[b]$	$\{M[b]\}$ - $kill[s]$	expressões contendo t
$M[a] \leftarrow b$	{ }	expressões da forma $M[x]$
if a > b goto L1 else goto L2	{ }	{ }
goto L	{ }	{ }
L :	{ }	{ }
$f(a1,...,an)$	{ }	expressões da forma $M[x]$
$t \leftarrow f(a1,...,an)$	{ }	expressões contendo t e expressões da forma $M[x]$

Fonte: Adaptado de (APPEL; PALSBERG, 2003)

$b \oplus c - kill[s]$ trata disto.

Uma instrução ($M[a] \leftarrow b$) pode modificar qualquer local de memória, então mata qualquer expressão de busca ($M[x]$). Se houver a certeza de que $a \neq x$, pode-se ser conservador e dizer que $M[a] \leftarrow b$ não mata $M[x]$. Isso é chamado análise de alias, que veremos na próxima seção.

Dados *gen* e *kill*, nós computamos também outras duas equações: *in* e *out*. Tais equações geram um conjunto de definições que atingem a entrada e a saída de cada nó do grafo de fluxo. Assim, as equações para *in* e *out* são:

$$in[n] = \bigcap_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

As equações podem ser resolvidas por iteração, iniciando, para todos os nós, com os conjuntos *out* vazios e com os conjuntos *in* completamente cheios (i.e., contendo todas as expressões possíveis). A justificativa é que o operador de interseção faz conjuntos menores, não maiores como o operador de união. Este algoritmo encontra o maior ponto fixo das equações.

É possível utilizar a abordagem vista na análise anterior para a verificação de *aliasing* entre ponteiros. Esta análise é denominada análise de ponteiros e será apresentada na próxima seção.

4.2 Análise de Ponteiros

Utilizamos análise de ponteiros, um tipo de análise de fluxo de dados, para saber mais sobre nomes diferentes que podem apontar para os mesmos locais de memória. O resultado da análise de ponteiros é uma relação $may-alias$: $pmay-aliasq$ se, em alguma execução do programa, p e q podem apontar para os mesmos dados. Tal como acontece com a maioria das análises de fluxo de dados, a informação estática não pode ser completamente precisa, então a relação entre $may-alias$ é conservadora: dizemos que $pmay-aliasq$ se não podemos provar que p nunca é um alias para q. O código 10 mostra a não-ocorrência de *aliasing* entre ponteiros.

Código 10 – Não-ocorrência de *aliasing*

```
void dontMakeAliasing(){
   int *p, *q;
   int h,i;
   p = &h;
   q = &i;
   *p = 0;
   *q = 5;
   a = *p;
}
```

Ao realizar uma análise de ponteiros ($may-alias$ neste código, seria possível afirmar que, no método `dontMakeAliasing` não há a ocorrência de *aliasing*, entre os ponteiros `p` e `q`, visto que os dois não apontam para a mesma região de memória.

No código 10, `p` e `q` são do mesmo tipo e sabemos que não podem estar em *aliasing*, então um deve ser 0. Para verificar distinções automaticamente, consideraremos uma classe de alias para cada ponto de criação. Ou seja, para cada declaração diferente em que um registro é alocado (isto é, para cada chamada para `malloc` em C ou `new` em Pascal ou Java), formamos uma nova classe de alias. Além disso, cada variável local ou global diferente cujo endereço é tomado é uma classe de alias. Um ponteiro (ou parâmetro de chamada por referência) pode apontar para variáveis de mais de uma classe de alias.

A fim de demonstrar como ocorre o processo de análise de ponteiros, utilizaremos o algoritmo proposto em (APPEL; PALSBERG, 2003).

O algoritmo de fluxo de dados manipula conjuntos de tuplas da forma (t,d,k) onde t é uma variável e d,k é a classe de alias de todas as instâncias do k-ésimo campo de um registro alocado na localização d. O conjunto em $[s]$ contém (t,d,k) se $t-k$ pode apontar para um registro de alias classe d no início da instrução s.

Trocaremos os conjuntos *gen* e *kill*, por uma função de transferência: dizemos que se A é a informação de alias (conjunto de tuplas) na entrada em uma declaração s, então $trans(A)$ é a

informação de *alias* na saída. A função de transferência é definida pela tabela 3 para os diferentes tipos de quadruplas.

Tabela 3 – Função de transferência para análise de ponteiros.

Estado s	$trans_s(s)$
$t \leftarrow b$	$(A - \sum_t) \cup \{(t,d,k) \mid (b,d,k) \in A\}$
$t \leftarrow b + k$	$(A - \sum_t) \cup \{(t,d,i) \mid (b,d,i-k) \in A\}$
$t \leftarrow b \oplus c$	$(A - \sum_t) \cup \{(t,d,i) \mid (b,d,j) \in A \vee (c,d,k) \in A\}$
$t \leftarrow M[b]$	$A \cup \sum_t$
$M[a] \leftarrow b$	A
if a > b goto L_1 else goto L_2	A
goto L	A
L :	A
$f(a1,...,an)$	A
$d : t \leftarrow allowRecord(a)$	$(A - \sum_t) \cup \{(t,d,0)\}$
$t \leftarrow f(a1,...,an)$	$A \cup \sum_t$

Fonte: Adaptado de (APPEL; PALSBERG, 2003)

Usamos a abreviatura $\sum_t$ para indicar o conjunto de todas as tuplas (t,d,k), onde d,k é a classe alias de qualquer campo de registro cujo tipo é compatível com a variável t.

As equações que representam as funções para análise de ponteiros são:

$$in[s0] = A_0$$
$$in[n] = \bigcup_{p \in pred[n]} out[p]$$
$$out[n] = trans_n(in[n])$$

Utilizando os conjuntos *in*, *out* e o algoritmo apresentado, dizemos que $pmay - aliasq$ no estado s se existe d, k tal que $(p,d,k) \in in[s]$ e $(q,d,k) \in in[s]$.

Na próxima seção, apresentaremos uma ferramenta que implementa análise de ponteiros, automatizando o processo de descoberta de *aliasing*.

4.3 WALA

O WALA (*T.J. Watson Libraries for Analysis*) é um conjunto de bibliotecas para análise estática e dinâmica de programas. Inicialmente foi desenvolvida pelo centro de pesquisas T.J. Watson, da IBM, mas em 2006, o projeto foi doado à comunidade. Está vinculada à licença publica do Eclipse (*Eclipse Public License*).

A biblioteca fornece um amplo leque de recursos, dentre os quais podemos destacar:

- Análise de Ponteiros / Construção do grafo de fluxo.

- *Framework* de análise de fluxo de dados intraprocedural.
- *Frontends* para múltiplas linguagens.
- Utilitários para análises genéricas / Estruturas de dados.

Originalmente, o WALA realiza as análises sobre os *bytecodes* gerados na compilação das classes, porém há um módulo exclusivo que trabalha com o Eclipse JDT. Desta forma, é possível integrar a ferramenta a outros *plugins* já desenvolvidos.

Para realizar alguma análise, é necessário, primeiramente, construir o grafo de fluxo do programa. A partir deste grafo de fluxo, é possível realizar até mais de uma análise. O código 11 apresenta um exemplo de como realizar uma análise com o WALA.

Código 11 – Exemplo de utilização do WALA

```
buildCG(String jarFile) {
   AnalyisScope scope = AnalysisScopeReader (1)
 . makeJavaBinaryAnalysisScope(jarFile, null);
   IClassHierarchy cha = ClassHierarchy.make(scope); (2)
   Iterable<Entrypoint> e = (3)
   Util.makeMainEntrypoints(scope, cha);
   AnalysisOptions o = new AnalysisOptions(scope, e); (4)
   CallGraphBuilder builder = (5)
   Util.makeZeroCFABuilder(o, new AnalysisCache(), cha, scope);
   CallGraph cg = builder.makeCallGraph(o, null); (6)
}
```

No exemplo acima, percebe-se que a análise (`scope`) é realizada sobre um arquivo já compilado (*jar*) em `(2)`. Logo após, o grafo de fluxo é construído em `(5)` e `(6)`, aplicando o tipo de análise escolhida. Ao utilizar o pacote com.ibm.wala.ide.jdt, é fornecida uma tradução da representação usada no Eclipse JDT para a representação utilizada nas análises do WALA.

Na próxima seção, veremos como o WALA poderia ser utilizado para análise do código e, assim, reduzir as assertivas geradas pelo método proposto em (SANTOS; LUCERO, 2017).

4.4 Redução das Obrigações de Prova

O módulo de análise de ponteiros implementado para a representação presente nos *plugins* que utilizam o Eclipse JDT recebe um projeto Java como parâmetro e verifica em quais pontos, neste projeto, pode haver *aliasing*.

Desta forma, utilizaremos este módulo para propor uma abordagem para a redução das obrigações de prova geradas no método apresentado no capítulo anterior.

Após a realização da análise, por meio do WALA, obteremos um conjunto de nós onde pode haver a possibilidade de *aliasing*. Ou seja, faremos uma modificação no método `doAssignmentAssertions()` para que ele verifique se, no momento da criação da assertiva, o nó avaliado pode conter *aliasing*. As modificações podem ser vistas nas partes sublinhadas.

Código 12 – Redução de assertivas sobre os campos

```
public void doAssignmentAssertions(List<ASTNode> mayNodes){
...
   for(Assignment assignment : assignmentVisitor.getAssignments()){
      Expression exp = assignment.getLeftHandSide();
        if(exp.getNodeType() == ASTNode.QUALIFIED_NAME && mayNodes.contains(exp){
         Name name = exp.getQualifier();
            SimpleName simpleName = exp.getName();
            for(QualifiedName qualified : fields){
            if(qualified.getName() == simpleName){
               AssertionStatement assertStt = ast.newAssertStatement();
                  InfixExpression infixExp = ast.newInfixExpression();
                    infixExp.setOperator(InfixExpression.Operator.NOT_EQUALS);
                    infixExp.setLeftOperand(name);
                    infixExp.setRightOperand(qualified.getQualifier());
               assertStt.setExpression(infixExp);
                    un.insertBefore(assertStt, assignment);
                }
            }
      }
   }
...
}
```

Basicamente, verificamos no código 12, se o campo para o qual iremos incluir as assertivas se encontra na lista de nós retornados pela análise de ponteiros, que podem conter *aliasing*. Caso isso ocorra, a assertiva é criada normalmente.

De maneira análoga, faremos a verificação no método `doMethodAssertions`. Porém, neste método, a verificação é dupla, pois além das assertivas são também criadas variáveis fantasmas.

Aproveitando-se da implementação do método `doMethodAssertions()` explanada anteriormente, adicionamos no código 13 uma restrição em cada geração de novos nós. Na primeira etapa para a criação de variáveis fantasmas, visando salvar o valor dos campos e na segunda, para a criação das assertivas propriamente ditas.

Utilizando tal abordagem, é possível reduzir consideravelmente o número de obrigações de prova gerada, tornando a operação mais prática e livrando o desenvolvedor da necessidade de ter que provar muitas assertivas.

Código 13 – Redução de assertivas sobre as chamadas a métodos

```
public void doMethodAssertions(List<ASTNode> mayNodes){
...
   for(QualifiedName qualified : fields){
      if(mayNodes.contains(qualified)){
         VariableDeclarationFragment vd = ast.newVariableDeclarationFragment();
         Name name = ast.newName(convertToString(qualified.getQualifier()));
         SimpleName simpleName =
             ast.newSimpleName(convertToString(qualified.getName()));
         QualifiedName initializer = ast.newQualifiedName(name, simpleName);
         SimpleName varName = ast.newSimpleName("t"+ i);
         vd.setName(varName);
         vd.setInitializer(initializer);
         VariableDeclarationStatement vds =
             ast.newVariableDeclarationStatement(vd);
         listRewrite.insertBefore(vds, nodeSelected, null);
         i++;
        }
   }
   for(MethodInvocation m : methods){
      i=1;
      for(QualifiedName qualified : fields){
            if(mayNodes.contains(qualified)){
               Name name =
                   ast.newName(convertToString(qualified.getQualifier()));
               SimpleName simpleName =
                   ast.newSimpleName(convertToString(qualified.getName()));
               QualifiedName left = ast.newQualifiedName(name, simpleName);
            SimpleName rightExp = ast.newSimpleName("t" + i);
            InfixExpression infixExpression = ast.newInfixExpression();
                infixExpression.setOperator(InfixExpression.Operator.EQUALS);
            infixExpression.setLeftOperand(left);
            infixExpression.setRightOperand(rightExp);
            AssertStatement as = ast.newAssertStatement();
            as.setExpression(infixExpression);
            listRewrite.insertAfter(as, m, null);
            i++;
            }
        }
   }
...
}
```

5
Conclusão

Sabendo que nem todas as refatorações fornecidas por ferramentas de desenvolvimento são corretas, propomos, neste trabalho, uma estratégia para a implementação de refatorações seguras para a plataforma Eclipse. Sugerimos que as implementações sejam baseadas em refatorações formais e que, para os casos em que as refatorações dependam de condições dinâmicas, se instrumentem os programas refatorados com obrigações de provas inseridas como assertivas. Para reduzir ou eliminar obrigações de prova, técnicas de análise de fluxo de dados poderão ser utilizadas.

Tomando como referência o catálogo de refatorações de Fowler (FOWLER et al., 1999), avaliamos o escopo das ferramentas Netbeans, Eclipse e Intellij Idea quanto a automação e segurança das refatorações.

A partir do trabalho de Santos e Lucero (2017), mostrou-se que, a refatoração *Inline Temp* automatizada no Eclipse não é correta, assim como foram propostos métodos de resolução. Neste trabalho, realizamos a implementação destes métodos, obtendo uma implementação correta. A refatoração foi integrada a um *plugin* que se utiliza do Eclipse JDT, seguindo o *framework* de refatorações do Eclipse.

O estudo sobre análise de fluxo de dados veio a incrementar o trabalho já realizado, abordando técnicas de análise usualmente exploradas em otimizações de código. Com o intuito de encontrar pontos em que possa ocorrer *aliasing*, propomos a utilização da análise de ponteiros implementada na biblioteca WALA visando a redução de assertivas geradas pelo nosso método.

Um ponto problemático encontrado ao utilizar o WALA foi a pouca documentação disponível por parte dos mantenedores, assim como eventuais *bugs* presentes na plataforma. Com base no tempo necessário para a compreensão acerca da ferramenta, apenas realizou-se um estudo inicial. Como um dos trabalhos futuros, propõe-se a complementação da refatoração *Inline Temp* segura com a implementação da integração com o WALA, fazendo uso mais efetivo de informações de *aliasing* para evitar a geração de mais assertivas. Este trabalho mostra que a

integração pode ser conveniente, desde que os dois projetos sejam bem integrados.

Propõe-se também, como trabalho futuro, a aplicação da estratégia apresentada neste trabalho para a implementação de outras refatorações que requerem a verificação de condições dinâmicas e que, usualmente, ferramentas como o Eclipse não verificam. Como exemplo, podemos citar a refatoração *move field*, que permite mover um campo de uma classe para outra.

De modo geral, existe uma estreita relação entre refatorações e transformações realizadas por compiladores. Particularmente, otimizações que visam a paralelização de código sequencial têm-se mostrado convenientes também como refatorações, ou seja, como transformações que visam a melhoria de código (KJOLSTAD; DIG; SNIR, 2010). Como trabalho futuro propõe-se também a implementação de refatorações para programas paralelos, utilizando otimizações para a resolução dos problemas relacionados a *aliasing*. Em especial refatorações relevantes não somente para paralelizar, mas para estruturar código, como por exemplo a *Split Loop* apresentada em (KJOLSTAD; DIG; SNIR, 2010).

Referências

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. [s.n.], 2006. 1000 p. ISBN 0321486811. Disponível em: <http://www.amazon.com/Compilers-Principles-Techniques-Tools-2nd/dp/0321486811>. Citado 2 vezes nas páginas 35 e 36.

APPEL, A. W.; PALSBERG, J. *Modern Compiler Implementation in Java*. 2nd. ed. New York, NY, USA: Cambridge University Press, 2003. ISBN 052182060X. Citado 4 vezes nas páginas 35, 37, 38 e 39.

BODDEN, E. Inter-procedural data-flow analysis with ifds/ide and soot. In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*. New York, NY, USA: ACM, 2012. (SOAP '12), p. 3–8. ISBN 978-1-4503-1490-9. Disponível em: <http://doi.acm.org/10.1145/2259051.2259052>. Citado na página 16.

BORBA, P. et al. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, v. 52, n. 1-3, p. 53–100, 2004. ISSN 01676423. Citado 4 vezes nas páginas 16, 24, 26 e 27.

CORNÉLIO, M.; CAVALCANTI, A.; SAMPAIO, A. Sound refactorings. *Science of Computer Programming*, v. 75, n. 3, p. 106–133, 2010. Citado 2 vezes nas páginas 16 e 24.

ECLIPSE. Eclipse Documentation. 2017. <http://http://help.eclipse.org> [Accessed: 10/07/2017]. Citado 2 vezes nas páginas 21 e 23.

FOWLER, M. et al. *Refactoring: Improving the Design of Existing Code*. [S.l.: s.n.], 1999. 1–337 p. ISSN 14359456. ISBN 9780201485677. Citado 5 vezes nas páginas 15, 18, 20, 27 e 43.

FUHRER, R. M.; KELLER, M.; KIEīUN, A. Advanced Refactoring in the Eclipse JDT : Past , Present , and Future. *Workshop on Refactoring Tools, WRT 2007, in conjunction with 21st European Conference on Object-Oriented Programming*, p. 30–31, 2007. ISSN 1436-9915. Citado na página 15.

HOARE, C. a. R. et al. Laws of programming. *Communications of the ACM*, v. 30, n. 8, p. 672–686, 1987. ISSN 00010782. Citado 3 vezes nas páginas 16, 25 e 26.

ISHIZAKI, K.; DAIJAVAD, S.; NAKATANI, T. Refactoring java programs using concurrent libraries. In: *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. New York, NY, USA: ACM, 2011. (PADTAD '11), p. 35–44. ISBN 978-1-4503-0809-0. Disponível em: <http://doi.acm.org/10.1145/2002962.2002970>. Citado na página 16.

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. A field study of refactoring challenges and benefits. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, p. 1, 2012. Disponível em: <http://dl.acm.org/citation.cfm?doid=2393596.2393655>. Citado na página 20.

KJOLSTAD, F.; DIG, D.; SNIR, M. Bringing the HPC Programmer's IDE into the 21st Century through Refactoring. *SPLASH Workshop on Concurrency for the Application Programmer*, p. 1–4, 2010. Citado na página 44.

LUCERO, G. Algebraic Laws for Object Oriented Programming with References. 2015. Citado 4 vezes nas páginas 16, 24, 27 e 28.

LUCERO, G.; NAUMANN, D.; SAMPAIO, A. Laws of programming for references. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 8301 LNCS, p. 124–139, 2013. ISSN 03029743. Citado na página 24.

MADHAVAN, R.; KOMONDOOR, R. Null dereference verification via over-approximated weakest pre-conditions analysis. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 46, n. 10, p. 1033–1052, out. 2011. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/2076021.2048144>. Citado na página 16.

PINTO, G. H.; KAMEI, F. What Programmers Say About Refactoring Tools?: An Empirical Investigation of Stack Overflow. *Proceedings of the 2013 ACM Workshop on Refactoring Tools*, p. 33–36, 2013. Citado na página 20.

PRESSMAN, R. S. *Software Engineering A Practitioner's Approach 7th Ed - Roger S. Pressman*. [S.l.: s.n.], 2009. 0 p. ISSN 1098-6596. ISBN 978-0-07-337597-7. Citado na página 15.

SANTOS, I. N.; LUCERO, G. Refatorações Corretas de Programas Java Implementadas em Eclipse. *Relatório final de pesquisa PIBIC/UFS*, 2017. Citado 6 vezes nas páginas 16, 24, 27, 28, 40 e 43.

Apêndices

APÊNDICE A – Levantamento de Refatorações em Catálogos e Ferramentas Automáticas

Tabela 4 – Comparativo entre catálogos e ferramentas automáticas em relação a refatorações.

Refatoração	**Fowler**	**Lucero**	**Kjolstad**	**Eclipse**	**Netbeans**	**Intellij**
Acrescentar Parâmetro	Sim	Sim	Não	Não	Não	Não
Auto-Encapsular Campo	Sim	Sim	Não	Não	Não	Não
Bloquear Loop	Não	Não	Sim	Não	Não	Não
Condensar Hierarquia	Sim	Não	Não	Não	Não	Não
Consolidar Expressão Condicional	Sim	Não	Não	Não	Não	Não
Consolidar Fragmentos Condicionais Duplicados	Sim	Não	Não	Não	Não	Não
Converter Projeto Procedural em Objetos	Sim	Não	Não	Não	Não	Não
Criar um Método Padrão	Sim	Não	Não	Não	Não	Não
Decompor Condicional	Sim	Não	Não	Não	Não	Não
Descer Campo na Hierarquia	Sim	Não	Não	Sim	Sim	Sim
Descer Método na Hierarquia	Sim	Não	Não	Som	Sim	Sim
Desembaraçar Herança	Sim	Não	Não	Não	Não	Não
Desenrolar Loop	Não	Não	Sim	Não	Não	Não
Dividir Computação em Comunicações Dependentes e Independentes	Não	Não	Sim	Não	Não	Não
Dividir Loop	Não	Não	Sim	Não	Não	Não
Dividir Struct em Campos	Não	Não	Sim	Não	Não	Não
Dividir Variável Temporária	Sim	Sim	Não	Não	Não	Não
Duplicar Dados Observados	Sim	Não	Não	Não	Não	Não
Encapsular Campo	Sim	Não	Não	Sim	Sim	Sim
Encapsular Coleção	Sim	Não	Não	Não	Não	Não
Encapsular Downcast	Sim	Não	Não	Não	Não	Não
Extrair Classe	Sim	Não	Não	Sim	Sim	Sim
Extrair Hierarquia	Sim	Não	Não	Não	Não	Não
Extrair Interface	Sim	Não	Não	Sim	Sim	Sim
Extrair Método	Sim	Sim	Não	Sim	Sim	Sim
Extrair Subclasse	Sim	Não	Não	Não	Não	Não
Extrair Superclasse	Sim	Não	Não	Sim	Sim	Sim
Fazer Comunicação Assíncrona	Não	Não	Sim	Não	Não	Não
Internalizar Classe	Sim	Não	Não	Não	Sim	Sim

Internalizar Método	Sim	Sim	Não	Sim	Sim	Sim
Internalizar Variável Temporária	Sim	Sim	Não	Sim	Sim	Sim
Introduzir Asserção	Sim	Não	Não	Não	Não	Não
Introduzir Extensão Local	Sim	Não	Não	Não	Sim	Não
Introduzir Método Externo	Sim	Não	Não	Não	Sim	Não
Introduzir Objeto Nulo	Sim	Não	Não	Não	Não	Não
Introduzir Objeto Parâmetro	Sim	Não	Não	Sim	Sim	Sim
Introduzir Variável Explicativa	Sim	Sim	Não	Não	Sim	Não
Mover Campo	Sim	Sim	Não	Sim	Sim	Sim
Mover Método	Sim	Sim	Não	Sim	Sim	Sim
Mudar de Referência para Valor	Sim	Não	Não	Não	Não	Não
Mudar de Valor para Referência	Sim	Não	Não	Não	Não	Não
Ocultar Delegação	Sim	Não	Não	Não	Não	Não
Ocultar Método	Sim	Não	Não	Sim	Sim	Não
Organize Block as Load/Compute/Store	Não	Não	Sim	Não	Não	Não
Parametrizar Método	Sim	Não	Não	Não	Não	Não
Preservar o Objeto Inteiro	Sim	Não	Não	Não	Não	Não
Remover Atribuições a Parâmetros	Sim	Sim	Não	Não	Não	Não
Remover Barreira com Sincronização Local	Não	Não	Sim	Não	Não	Não
Remover Barreiras Superficiais	Não	Não	Sim	Não	Não	Não
Remover Flag de Controle	Sim	Não	Não	Não	Não	Não
Remover Intermediário	Sim	Não	Não	Não	Não	Não
Remover Método de Gravação	Sim	Não	Não	Não	Não	Não
Remover Parâmetro	Sim	Não	Não	Não	Não	Não
Renomear	Sim	Não	Não	Sim	Sim	Sim
Restringir Ponteiro	Não	Não	Sim	Não	Não	Não
Reunir Dados	Não	Não	Sim	Não	Não	Não
Separar a Consulta do Modificador	Sim	Não	Não	Não	Não	Não
Separar o Domínio da Apresentação	Sim	Não	Não	Não	Não	Não
Subir Campo na Hierarquia	Sim	Não	Não	Sim	Sim	Sim
Subir Método na Hierarquia	Sim	Não	Não	Sim	Sim	Sim
Subir o Corpo do Construtor na Hierarquia	Sim	Não	Não	Não	Sim	Sim
Substituir Atributo por Objeto	Sim	Não	Não	Não	Não	Não
Substituir Código de Erro por Exceção	Sim	Não	Não	Não	Não	Não
Substituir Comando Condicional por Polimorfismo	Sim	Não	Não	Não	Não	Não
Substituir Condição Aninhada por Cláusulas Guarda	Sim	Não	Não	Não	Não	Não
Substituir Delegação por Herança	Sim	Não	Não	Não	Não	Não
Substituir Enumeração pelo Padrão State/Strategy	Sim	Não	Não	Não	Não	Não

Substituir Enumeração por Classe	Sim	Não	Não	Não	Não	Não
Substituir Enumeração por Subclasses	Sim	Não	Não	Não	Não	Não
Substituir Exceção por Teste	Sim	Não	Não	Não	Não	Não
Substituir Herança por Delegação	Sim	Não	Não	Não	Não	Não
Substituir Método por Objeto Método	Sim	Sim	Não	Não	Não	Não
Substituir Números Mágicos por Constantes Simbólicas	Sim	Não	Não	Não	Não	Não
Substituir o Algoritmo	Sim	Não	Não	Não	Não	Não
Substituir o Construtor por um Método Fábrica	Sim	Não	Não	Sim	Sim	Sim
Substituir Parâmetro por Método	Sim	Não	Não	Não	Não	Não
Substituir Parâmetro por Métodos Explícitos	Sim	Não	Não	Não	Não	Não
Substituir Registro por Classe de Dados	Sim	Não	Não	Não	Não	Não
Substituir Subclasse por Campos	Sim	Não	Não	Não	Não	Não
Substituir Variável Temporária por Consulta	Sim	Sim	Não	Não	Não	Sim
Substituir Vetor por Objeto	Sim	Não	Não	Não	Não	Não
Transformar Associação Bidirecional em Unidirecional	Sim	Não	Não	Não	Não	Não
Transformar Associação Unidirecional em Bidirecional	Sim	Não	Não	Não	Não	Não

APÊNDICE B – Código-fonte da Refatoração *Inline Temp* correta

B.1 Classe RefactoringHandler

Código 14 – Classe que dispara a refatoração

```
@SuppressWarnings("restriction")
public class RefactoringHandler extends AbstractHandler {
   @Override
   public Object execute(ExecutionEvent event) throws ExecutionException {
      doInlineTemp();
      return null;
   }
   public void doInlineTemp(){
      IWorkbenchPage page = PlatformUI.getWorkbench().getActiveWorkbenchWindow()
        .getActivePage();
      ITextEditor editor = (ITextEditor) page.getActiveEditor();
      IJavaElement elem =
          JavaUI.getEditorInputJavaElement(editor.getEditorInput());
      ICompilationUnit un = (ICompilationUnit) elem;
      ISelectionService ss =
          PlatformUI.getWorkbench().getActiveWorkbenchWindow().
        getSelectionService();
      ITextSelection its = (ITextSelection) ss.getSelection();
      try {
         Shell shell =
             PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell();
         SoundInlineTempRefactoring iir = new SoundInlineTempRefactoring(un,
             its.getOffset(), its.getLength());
         RefactoringWizard rw = new InlineTempWizard(iir);
         RefactoringStarter rs = new RefactoringStarter();
         rs.activate(rw, shell, "Sound Inline Temp", 1);
         ge.perform(monitor);
      } catch (Exception e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
      }
   }
}
```

B.2 Classe SoundInlineTemp

Código 15 – Classe que realiza as checagens e, logo após, a refatoração Inline Temp

```
@SuppressWarnings("restriction")
public class SoundInlineTempRefactoring extends InlineTempRefactoring{
   private int fSelectionStart;
   private int fSelectionLength;
   private ICompilationUnit fCu;
   private CompilationUnit fASTRoot;
   private VariableDeclaration fVariableDeclaration;
   private boolean fModifications;
   private int nodePosition;
   Expression right;
   List<QualifiedName> fields = new ArrayList<QualifiedName>();
   ASTRewrite rewrite;
   ListRewrite listRewrite;
   public SoundInlineTempRefactoring(ICompilationUnit unit, int selectionStart,
       int selectionLength){
      super(unit, null, selectionStart, selectionLength);
      fCu = unit;
      fSelectionLength = selectionLength;
      fSelectionStart = selectionStart;
      fVariableDeclaration = null;
      fASTRoot = null;
      fModifications = false;
   }
   @Override
   public Change createChange(IProgressMonitor pm) throws CoreException {
      Change change = super.createChange(pm);
      change.perform(pm);
      System.out.println(fCu.getSource());
      if(fModifications) {
         doAssignmentAssertions();
         doMethodAssertions();
         String source = fCu.getSource();
           Document document = new Document(source);
         TextEdit edits = rewrite.rewriteAST(document, null);
         try {
            edits.apply(document);
          } catch (MalformedTreeException es) {
```

```
            // TODO Auto-generated catch block
            es.printStackTrace();
        } catch (BadLocationException es) {
            // TODO Auto-generated catch block
            es.printStackTrace();
        }
      String newSource = document.get();
      // update of the compilation unit
      fCu.getBuffer().setContents(newSource);
      System.out.println(rewrite.toString());
   }
   return change;
}

@Override
public RefactoringStatus checkInitialConditions(IProgressMonitor pm) throws
    CoreException {
   CompilationUnit parse = parse(fCu);
   ASTNode nodeSelected = NodeFinder.perform(parse, this.fSelectionStart,
       this.fSelectionLength);
   if(checkReadOnly(parse))
      return RefactoringStatus.createFatalErrorStatus("One or most variables
          changed after expression");
   else {
      int totalAsserts = checkAssignmentAssertions(nodeSelected) +
          checkMethodAssertions();
      if(totalAsserts > 0) {
         fModifications = true;
         RefactoringStatus status = super.checkInitialConditions(pm);
         status.addInfo(totalAsserts + " asserts will be generated");
         return status;
      }
      return super.checkInitialConditions(pm);
   }
}
public boolean checkReadOnly(CompilationUnit un) {
   boolean error = false;
   getVariableDeclaration();
   right = fVariableDeclaration.getInitializer();
   AssignmentVisitor assignmentVisitor = new AssignmentVisitor();
   SimpleNameVisitor simpleNameVisitor = new SimpleNameVisitor();
   QualifiedNameVisitor qualifiedNameVisitor = new QualifiedNameVisitor();
```

```
    right.accept(simpleNameVisitor);
    int expressionPosition = right.getStartPosition();
    List<SimpleName> vard = simpleNameVisitor.getMethods();
    right.accept(qualifiedNameVisitor);
    List<QualifiedName> fieldList = qualifiedNameVisitor.getMethods();
    MethodDeclaration methodVerified= getMethodDeclaration(un);
    List<ASTNode> statementsList = methodVerified.getBody().statements();
    int pos = fVariableDeclaration.getParent().getStartPosition();
    int i = 0;
    for(ASTNode node : statementsList) {
       if(node.getStartPosition() == pos)
          break;
       i++;
    }
    nodePosition = i;
    methodVerified.accept(assignmentVisitor);
    List<String> variablesList = new ArrayList<String>();
    for(Assignment assignment : assignmentVisitor.getMethods()){
       Expression aux = assignment.getLeftHandSide();
       int assignPosition = aux.getStartPosition();
       if(assignPosition > expressionPosition){
          if(aux.getNodeType() == 40){
             fields.add((QualifiedName)aux);
          }
          variablesList.add(convertToString(aux));
       }
    }
    for(SimpleName sn : vard){
       if(variablesList.contains(convertToString(sn))){
          error = true;
       }
      }
    for(QualifiedName qn : fieldList){
       if(variablesList.contains(convertToString(qn))){
          error = true;
       }
      }
    return error;
}
public int checkAssignmentAssertions(ASTNode node) {
    int asserts = 0;
      Expression right = fVariableDeclaration.getInitializer();
```

```
      QualifiedNameVisitor fa = new QualifiedNameVisitor();
      right.accept(fa);
      List<QualifiedName> lfa = fa.getMethods();
      if(lfa.size() > 0){
       for(QualifiedName atual : fields){
          SimpleName name = atual.getName();
          for(QualifiedName fac : lfa){
             SimpleName rn = fac.getName();
             if(convertToString(name).compareTo(convertToString(rn)) == 0)
                   asserts++;
          }
       }
      }
      return asserts;
  }
  public void doAssignmentAssertions() throws CoreException{
      QualifiedNameVisitor fa = new QualifiedNameVisitor();
      right.accept(fa);
      List<QualifiedName> lfa = fa.getMethods();
      if(lfa.size() > 0){
          ASTParser parser = ASTParser.newParser(AST.JLS3);
          parser.setSource(fCu);
          CompilationUnit cu = (CompilationUnit) parser.createAST(null);
          AST ast = cu.getAST();
       rewrite = ASTRewrite.create( ast );
          MethodDeclaration method = getMethodDeclaration(cu);
       Block b = method.getBody();
       listRewrite = rewrite.getListRewrite(b,
                   Block.STATEMENTS_PROPERTY);
       AssignmentVisitor assignVisitor = new AssignmentVisitor();
       b.accept(assignVisitor);
       for(QualifiedName atual : fields){
             SimpleName name = atual.getName();
             Name exp = atual.getQualifier();
             for(QualifiedName fac : lfa){
                SimpleName rn = fac.getName();
                if(convertToString(name).compareTo(convertToString(rn)) == 0){
                   Name re = fac.getQualifier();
                   SimpleName left = ast.newSimpleName(convertToString(re));
                   SimpleName rightExp =
                       ast.newSimpleName(convertToString(exp));
                   InfixExpression infixExpression = ast .newInfixExpression();
```

```
                infixExpression.setOperator(
                      InfixExpression.Operator.NOT_EQUALS);
                infixExpression.setLeftOperand(left);
                infixExpression.setRightOperand(rightExp);
                AssertStatement as = ast.newAssertStatement();
                as.setExpression(infixExpression);
                Assignment at = null;
                for(Assignment a : assignVisitor.getMethods()) {
                   if(convertToString(a.getLeftHandSide()).
                          compareTo(convertToString(atual)) == 0) {
                      at = a;
                      break;
                   }
                }
                listRewrite.insertBefore(as, at.getParent(), null);
             }
          }
      }
     }
}
public int checkMethodAssertions() {
   int asserts = 0;
   CompilationUnit cu = parse(fCu);
     Expression right = fVariableDeclaration.getInitializer();
     MethodDeclaration method = getMethodDeclaration(cu);
     QualifiedNameVisitor fa = new QualifiedNameVisitor();
     MethodInvocationVisitor miv = new MethodInvocationVisitor();
     right.accept(fa);
     method.accept(miv);
     List<QualifiedName> lfa = fa.getMethods();
     List<MethodInvocation> lmi = miv.getMethods();
     if(lfa.size() > 0 && lmi.size() > 0){
      for (int i = 0; i < lmi.size(); i++) {
         for (int j = 0; j < lfa.size(); j++) {
            asserts++;
         }
      }
     }
     return asserts;
}
public void doMethodAssertions() throws JavaModelException{
     ASTParser parser = ASTParser.newParser(AST.JLS3);
```

```
parser.setSource(fCu);
CompilationUnit cu = (CompilationUnit) parser.createAST(null);
AST ast = cu.getAST();
MethodDeclaration method = getMethodDeclaration(cu);
QualifiedNameVisitor fa = new QualifiedNameVisitor();
MethodInvocationVisitor miv = new MethodInvocationVisitor();
right.accept(fa);
method.accept(miv);
List<QualifiedName> lfa = fa.getMethods();
List<MethodInvocation> lmi = miv.getMethods();
if(lfa.size() > 0 && lmi.size() > 0){
 int i=1;
 for(QualifiedName qn : lfa){
    VariableDeclarationFragment vd =
        ast.newVariableDeclarationFragment();
    Name name = ast.newName(convertToString(qn.getQualifier()));
    SimpleName sn = ast.newSimpleName(convertToString(qn.getName()));
    QualifiedName e = ast.newQualifiedName(name, sn);
    SimpleName varName = ast.newSimpleName("t"+ i);
    vd.setName(varName);
    vd.setInitializer(e);
    VariableDeclarationStatement vds =
        ast.newVariableDeclarationStatement(vd);
    listRewrite.insertAt(vds,nodePosition, null);
    i++;
 }
 for(MethodInvocation m : lmi){
       i=1;
       for(QualifiedName qn : lfa){
          Name name = ast.newName(convertToString(qn.getQualifier()));
          SimpleName sn =
              ast.newSimpleName(convertToString(qn.getName()));
          QualifiedName left = ast.newQualifiedName(name, sn);
       SimpleName rightExp = ast.newSimpleName("t" + i);
       InfixExpression infixExpression = ast .newInfixExpression();
       infixExpression.setOperator(InfixExpression.Operator.EQUALS);
       infixExpression.setLeftOperand(left);
       infixExpression.setRightOperand(rightExp);
       AssertStatement as = ast.newAssertStatement();
       as.setExpression(infixExpression);
       List <Statement> statements = listRewrite.getRewrittenList();
       int pos = m.getParent().getStartPosition();
```

```
            Statement state = null;
            for(Statement node : statements) {
               if(node.getStartPosition() == pos) {
                  state = node;
               }
            }
            listRewrite.insertAfter(as, state, null);
            i++;
            }
      }
     }
}
@Override
public RefactoringStatus checkFinalConditions(IProgressMonitor pm) throws
    CoreException {
   return super.checkFinalConditions(pm);
}
private String convertToString(Object a){
   return a + "";
}
 private static CompilationUnit parse(ICompilationUnit unit) {
     ASTParser parser = ASTParser.newParser(AST.JLS3);
     parser.setKind(ASTParser.K_COMPILATION_UNIT);
     parser.setSource(unit);
     parser.setResolveBindings(true);
     return (CompilationUnit) parser.createAST(null);
 }
private CompilationUnit getASTRoot() {
   if (fASTRoot == null) {
      fASTRoot= RefactoringASTParser.parseWithASTProvider(fCu, true, null);
   }
   return fASTRoot;
}
public VariableDeclaration getVariableDeclaration() {
   if (fVariableDeclaration == null) {
      fVariableDeclaration=
          TempDeclarationFinder.findTempDeclaration(getASTRoot(),
          fSelectionStart, fSelectionLength);
   }
   return fVariableDeclaration;
}
public MethodDeclaration getMethodDeclaration(CompilationUnit un) {
```

```
        int pos = right.getStartPosition();
      MethodDeclarationVisitor mdv = new MethodDeclarationVisitor();
       un.accept(mdv);
      MethodDeclaration method = null;
        for(MethodDeclaration md : mdv.getMethods()){
         if(md.getStartPosition() < pos){
            method = md;
         }
         else{
            break;
         }
        }
        return method;
   }
}
```

B.3 Classe Visitor

Código 16 – Classe que realiza uma visita em um nó da árvore.

```
public class Visitor extends ASTVisitor{
   List<ASTNode> nodes = new ArrayList<ASTNode>();
   @Override
   public boolean visit(ASTNode node){
      nodes.add(node);
      return super.visit(node);
   }
   public List<ASTNode> getNodes(){
      return nodes;
   }
   public ASTNode getFirst(){
      return nodes.get[0];
   }
}
```